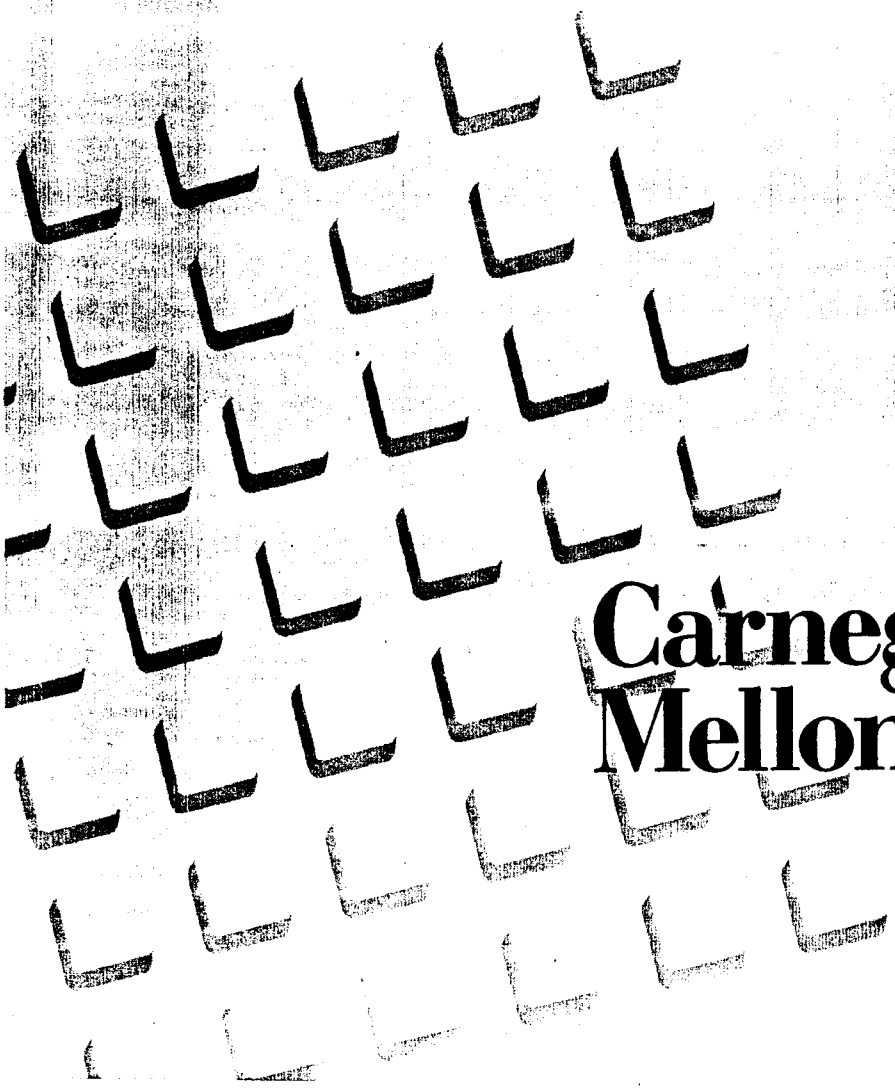
Beyond Objects: A Software Design Paradigm Based on Process Control

Mary Shaw*
January 1994
CMU-CS-94-154



Carnegie Mellon

Beyond Objects: A Software Design Paradigm Based on Process Control

Mary Shaw*
January 1994
CMU-CS-94-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

To appear in Software Engineering Notes, Vol. 20, No. 1, January 1995

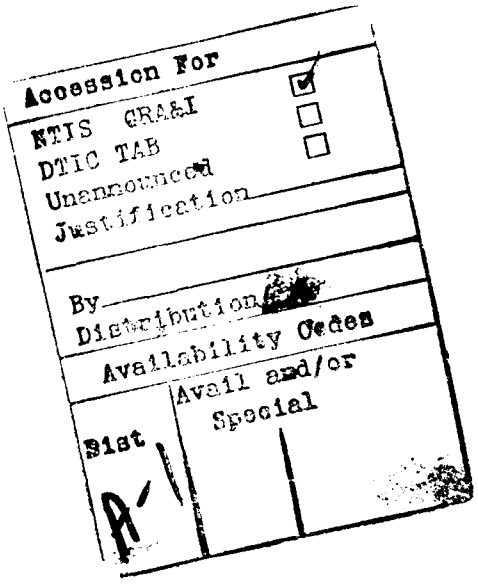
Also appears as CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-15, ESC-TR-94-015.

Abstract

A standard demonstration problem in object-oriented programming is the design of an automobile cruise control. This design exercise demonstrates object-oriented techniques well, but it does not ask whether the object-oriented paradigm is the best one for the task. Here we examine the alternative view that cruise control is essentially a control problem. We present a new software organization paradigm motivated by process control loops. The control view leads us to an architecture that is dominated by analysis of a classical feedback loop rather than by the identification of discrete stateful components to treat as objects. The change in architectural model calls attention to important questions about the cruise control task that aren't addressed in an object-oriented design.

*Also affiliated with the Software Engineering Institute, Carnegie Mellon University.

This research was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense) and by a grant from Corporate Research. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies either expressed or implied, of any of the sponsors.



Keywords: cruise control, software architecture, feedback control architecture, software design

Beyond Objects:
A Software Design Paradigm Based on Process Control

1. Design Idioms for Software Architectures

Explicit organization patterns, or idioms, increasingly guide the composition of modules into complete systems. This stage of the design is usually called the architecture, and a number of common patterns are in general, though quite informal, use [Garlan and Shaw 93]. One of these, the object-oriented architecture [Booch 86], is the subject of much current attention. Although several architectural idioms have strong advocates, no single paradigm dominates. The choice of an architecture should instead depend on the computational character of the application.

Here we explore a software idiom based on process control loops. This system organization is not widely recognized in the software community; nevertheless it seems to quietly appear within designs dominated by other models. Unlike object-oriented or functional design, which are characterized by the kinds of components that appear, control loop designs are characterized both by the kinds of components and the special relations that must hold among the components.

The paper first explains process control models and derives a software paradigm for control loop organizations. Then it applies the result to a well-known problem, the design of a cruise control system. The differences between the control-loop-based and the object-oriented designs reveal relative strengths of the models for problems of this kind. The control view clarifies the different roles played by various problem inputs; further, it helps the designer recognize a safety problem and a system limitation. Drawing on the knowledge of process control also offers prospects for design guidance and quantitative analysis of system response characteristics.

1.1. Process control paradigms

Continuous processes of many kinds convert input materials to products with specific properties by performing operations on the inputs and on intermediate products. The values of measurable properties of system state (materials, equipment settings, etc.) are called the *variables* of the process. Process variables that measure the output materials are called the *controlled variables* of the process. The properties of the input materials, intermediate products, and operations are captured in other process variables. In particular, the *manipulated variables* are associated with things that can be changed by the control system in order to regulate the process. Process variables must not be confused with program variables; this error can lead to disaster [Åström and Wittenmark 84, Leveson 86, Perry 84, Seborg et al 89].

Process Control Definitions

Process variables: properties of the process that can be measured; several specific kinds are often distinguished. **Do not** confuse process variables with program variables.

Controlled variable: process variable whose value the system is intended to control

Input variable: process variable that measures an input to the process

Manipulated variable: process variable whose value can be changed by the controller

Set point: the desired value for a controlled variable

Open loop system: system in which information about process variables is not used to adjust the system.

Closed loop system: system in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions.

Feedback control system: the controlled variable is measured and the result is used to manipulate one or more of the process variables

Feedforward control system: some of the process variables are measured and disturbances are compensated for without waiting for changes in the controlled variable to be visible.

The purpose of a control system is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values called the set points. If the input materials are pure, if the process is fully-defined, and if the operations are completely repeatable, the process can simply run without surveillance. Such a process is called an open loop system. Figure 1 shows such a system, a hot-air furnace that uses a constant burner setting to raise the temperature of the air that passes through. A similar furnace that uses a timer to turn the burner off and on at fixed intervals is also an open loop system.

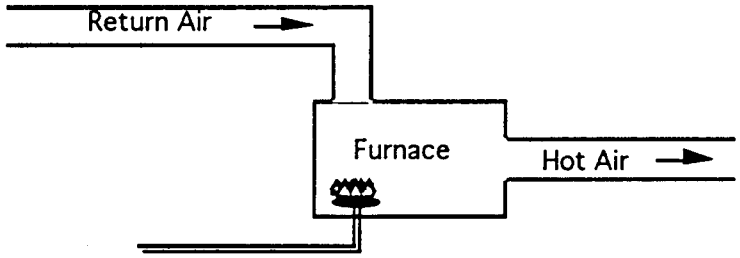


Figure 1: Open loop temperature control

The open-loop assumptions are rarely valid for physical processes in the real world. More often, properties such as temperature, pressure and flow rates are monitored, and their values are used to control the process by changing the settings of apparatus such as valves, heaters, and chillers. Such systems are called closed loop systems. A home thermostat is a common example: the air temperature at the thermostat is measured, and the furnace is turned on and off as necessary to maintain the desired temperature (the set point). Figure 2 shows the addition of a thermostat to convert Figure 1 to a closed-loop system.

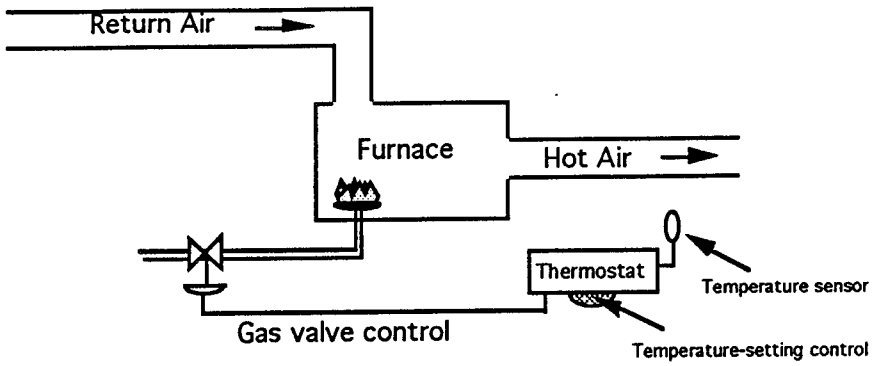


Figure 2: Closed loop temperature control

There are two general forms of closed loop control. *Feedback control*, illustrated in Figure 3, adjusts the process based on measurements of the controlled variable. The important components of a feedback controller are the process definition, the process variables (including designated input and controlled variables), a sensor to obtain the controlled variable from the physical output, the set point (target value for the controlled variable), and a control algorithm. Figure 2 corresponds to Figure 3 in the following ways: The furnace with burner is the process; the thermostat is the controller; the return air temperature is the input variable; the hot air temperature is the controlled variable; the thermostat setting is the set point; and the temperature sensor is the sensor.

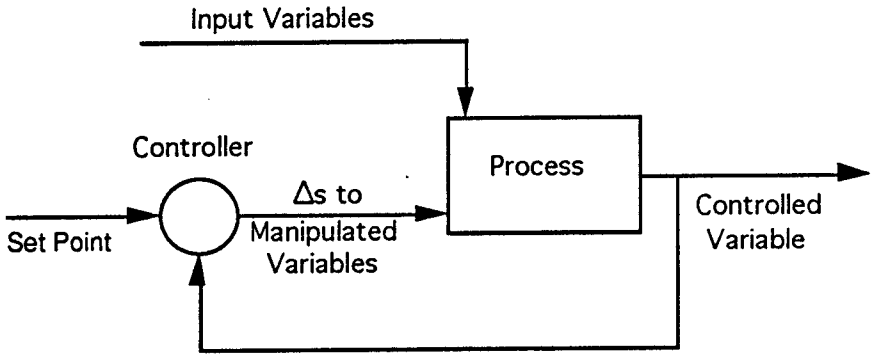


Figure 3: Feedback Control

Feedforward control, shown in Figure 2, anticipates future effects on the controlled variable by measuring other process variables whose values may be more timely; it adjusts the process based on these variables. The important components of a feedforward controller are essentially the same as for a feedback controller except that the sensor(s) obtain values of input or intermediate variables. It is valuable when lags in the process delay the effect of control changes.

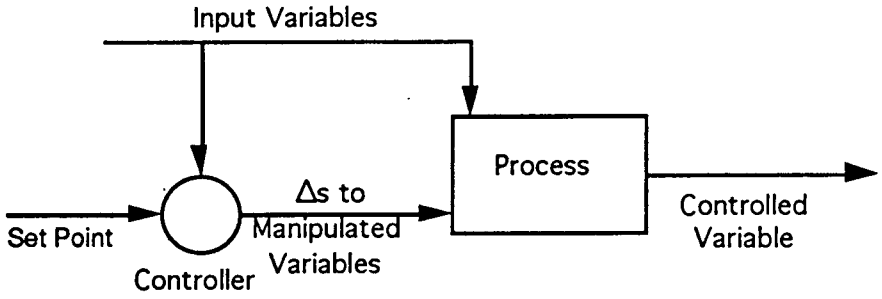


Figure 4: Feedforward Control

These are simplified models. They do not deal with complexities such as properties of sensors, transmission delays, and calibration issues. They ignore the response characteristics of the system such as gain, lag, and hysteresis. They don't show how to combine feedforward and feedback or choose which process variables to manipulate. Chemical engineering provides excellent quantitative models for predicting how processes will react to various control algorithms; indeed there are a number of standard strategies [Perry 84, Section 22]. These are mentioned in Section 3.4, but detailed discussion is beyond the scope of this paper.

1.2. Software paradigm for process control

We usually think of software as algorithmic: we compute outputs (or execute continuous systems) solely on the basis of the inputs. This normal model does not allow for external perturbations; if non-input values of a computation change spontaneously, this is regarded as a hardware error. The normal software model corresponds to an open loop system; in most cases it is entirely appropriate. However, when the operating conditions of a software system are not completely predictable—especially when the software is operating a physical system—the purely algorithmic model breaks down. When the execution of a software system is affected by external disturbances—forces or events that are not directly visible to or controllable by the software—this is an indication that a control paradigm should be considered for the software architecture.

We can now establish a paradigm for software that controls continuous processes. The elements of this pattern incorporate the essential parts of a process control loop, and the methodology requires explicit identification of these parts:

Computational elements: separate the process of interest from the control policy.

Process definition, including mechanisms for manipulating some process variables.

Control algorithm to decide how to manipulate process variables, including a model for how the process variables reflect the true state.

Data elements: continuously updated process variables and sensors that collect them.

Process variables, including designated input, controlled, and manipulated variables and knowledge of which can be sensed.

Set point, or reference value for controlled variable.

Sensors to obtain values of process variables pertinent to control.

The ***control loop paradigm*** establishes the relation that the control algorithm exercises surveillance: it collects information about the actual and intended states of the process and tunes the process variables to drive the actual state toward the intended state.

The two computational elements separate issues about desired functionality from issues about responses to external disturbances. For a software system, we can bundle the process and the process variables; that is we can regard the process definition together with the process variables and sensors as a single subsystem whose input and controlled variables are visible in the subsystem interface. We can then bundle the control algorithm and the set point as a second subsystem; this controller has continuous access to current values of the set point and the monitored variables; for a feedback system, this will be the controlled variable. There are two interactions between these major systems: the controller receives values of process variables from the process, and the controller supplies continuous guidance to the process about changes to the manipulated variables.

The result is a particular kind of dataflow architecture. The primary characteristic of dataflow architectures is that the components interact by providing data to each other, each component executing when data is available. Most dataflow architectures involve independent (often concurrent) processes and pacing that depends on the rates at which the processes provide data for each other. The control loop paradigm assumes further that data related to process variables is updated continuously.

Other, perhaps more familiar, members of the dataflow family are batch sequential processing and pipe-and-filter architectures. Both are largely linear: data enters the system and is processed progressively by a number of distinct computations. In batch sequential architectures each phase runs to completion and delivers the result (historically as a magnetic tape!) to the next. In pipe-and-filter architectures, on the other hand, each filter processes its input stream incrementally (in unix, by characters or lines) so the filters can operate concurrently, at least in principle. The control loop architecture described here differs from both by the commitment to a dataflow loop and in the intrinsic asymmetry of the control element from the process element.

It is appropriate to consider a control loop design when:

- the task involves continuing action, behavior, or state
- the software is *embedded*; that is, it controls a physical system
- uncontrolled, or open loop, computation does not suffice, usually because of external perturbations or imprecise knowledge of the external state

2. Cruise control

2.1. The cruise control problem

Disciplines often work out the details of their methods through *type problems*, common examples used by many different people to compare their models and methods [Shaw et al 94]. Booch and others have used the cruise control problem to explore the differences between object-oriented and functional (traditional procedural) programming [Atlee and Gannon 91, Booch 86, Ward 84]. As given by Booch, this problem is:

> A cruise control system exists to maintain the speed of a car, even over varying terrain. In Figure 5 we see the block diagram of the hardware for such a system.

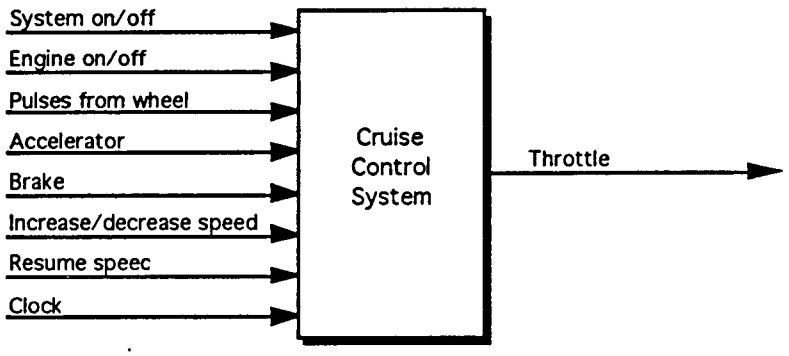


Figure 5: Booch block diagram for cruise control

There are several inputs:

- **System on/off** — If on, denotes that the cruise-control system should maintain the car speed.
- **Engine on/off** — If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
- **Pulses from wheel** — A pulse is sent for every revolution of the wheel.
- **Accelerator** — Indication of how far the accelerator has been pressed.
- **Brake** — On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
- **Increase/Decrease Speed** — Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
- **Resume** — Resume the last maintained speed; only applicable if the cruise-control system is on.
- **Clock** — Timing pulse every millisecond.

There is one output from the system:

- **Throttle** — Digital value for the engine throttle setting.

The problem does not clearly state the rules for deriving the output from the set of inputs. Booch provides a certain amount of elaboration in the form of a data flow diagram, but some questions remain unanswered. In the design below, missing details are supplied to match the apparent behavior of the cruise control on the author's car. Moreover, the inputs provide two kinds of information: whether the cruise control is active, and if so what speed it should maintain.

The problem statement says the output is a value for the engine **throttle** setting. In classical process control the corresponding signal would be a change in the throttle setting; this avoids calibration and wear problems with the sensors and engine. A more conventional cruise control requirement would thus specify control of the **current speed** of the vehicle. However, **current speed** is not explicit in the problem statement, though it does appear implicitly as "maintained speed" in the descriptions of some of the inputs. If the requirement addresses **current speed, throttle** setting remains an appropriate output from the control algorithm. To

avoid unnecessary changes in the problem we assume accurately calibrated digital control and achieve the effect of incremental signals by retaining the previous throttle value in the controller.

The problem statement also specifies a millisecond **clock**. In the object-oriented solution, the **clock** is used only in combination with the **wheel pulses** to determine the **current speed**. Presumably the process that computes the speed will count the number of clock pulses between wheel pulses. A typical automobile tire has a circumference of about 6 feet, so at 60 mph (88 ft/sec) there will be about 15 wheel pulses per second. The problem is overspecified in this respect: a slower clock or one that delivered current time on demand with sufficient precision would also work and would require less computing. Further, a single system clock is not required by the problem, though it might be convenient for other reasons.

These considerations lead to a restatement of the problem: *Whenever the system is active, determine the desired speed and control the engine throttle setting to maintain that speed.*

2.2. Object view of cruise control

Booch structures an object-oriented decomposition of the system around objects that exist in the task description. This yields a decomposition whose elements correspond to important quantities and physical entities in the system. The result appears in Figure 6, where the blobs represent objects and the directed lines represent dependencies among objects. Although the target speed did not appear explicitly in the problem statement, it does appear in Figure 6 as "Desired Speed".

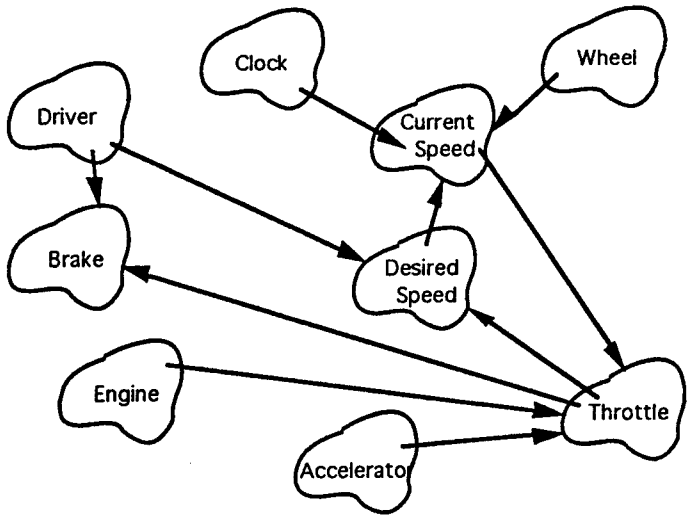


Figure 6: Booch's object-oriented design for cruise control

2.3. Process control view of cruise control

Section 1.2 suggests the selection of a control loop architecture when the software is embedded in a physical system that involves continuing behavior, especially when the system is subject to external perturbations. These conditions hold in the case of cruise control: the system is supposed to maintain constant speed in an automobile despite variations in terrain, vehicle load,

air resistance, fuel quality, etc. To develop a control loop architecture for this system, we begin by identifying the essential system elements as described in Section 1.2:

Computational elements

- *Process definition:* Since the cruise control software is driving a mechanical device (the engine), the details are not relevant. For our purposes, the process receives a **throttle** setting and turns the car's wheels. There may in fact be more computers involved, for example in controlling the fuel-injection system. From the standpoint of the cruise control subsystem, however, the process takes a throttle setting as input and controls the speed of the vehicle.
- *Control algorithm:* This algorithm models the **current speed** based on the **wheel pulses**, compares it to the **desired speed**, and changes the throttle setting. The **clock** input is needed to model **current speed** based on intervals between **wheel pulses**. Since the problem requires an exact **throttle** setting rather than a change, the current **throttle** setting must be maintained by the control algorithm. The policy decision about how much to change the **throttle** setting for a given discrepancy between **current speed** and **current speed** is localized in the control algorithm.

Data elements

- *Controlled variable:* For the cruise control, this is the **current speed** of the vehicle.
- *Manipulated variable:* For the cruise control, this is the **throttle** setting.
- *Set point:* The **desired speed** is set and modified by the **accelerator** input and the **increase/decrease speed** input, respectively. Several other inputs help control whether the cruise control is currently controlling the car: **System on/off**, **engine on/off**, **brake**, and **resume**. These interact: **resume** restores automatic control, but only if the entire system is on. These inputs are provided by the human driver (the operator, in process terms).
- *Sensor for controlled variable:* For cruise control, the current state is the **current speed**, which is modeled on data from a sensor that delivers **wheel pulses** using the **clock**. However, see discussion below about the accuracy of this model.

The restated control task was, "Whenever the system is active determine the desired speed and control the engine throttle setting to maintain that speed." Note that only the **current speed** output, the **wheel pulses** input, and the **throttle** manipulated variable are used outside the set point and active/inactive determination. This leads immediately to two subproblems: the interface with the driver, concerned with "whenever the system is active determine the desired speed" and the control loop, concerned with "control the engine throttle setting to maintain that speed."

The latter is the actual control problem; we'll examine it first. Figure 7 shows a suitable architecture for the control system. The first task is to model the **current speed** from the **wheel pulses**; the designer should validate this model carefully. The model could fail if the wheels spin; this could affect control in two ways. If the **wheel pulses** are being taken from a drive wheel and the wheel is spinning, the cruise control would keep the wheel spinning (at constant speed) even if the vehicle stops moving. Even worse, if the **wheel pulses** are being taken from a non-drive wheel and the drive wheels are spinning, the controller will be misled to believe that the current speed is too slow and will continually increase the **throttle** setting. The designer should also consider whether the controller has full control authority over the process. In the case of cruise control, the only manipulated variable is the **throttle**: the brake is not available.

As a result, if the automobile is coasting faster than the desired speed, the controller is powerless to slow it down.

The controller also receives two inputs from the set point computation: the **active/inactive toggle**, which indicates whether the controller is in charge of the **throttle**, and the **desired speed**, which only needs to be valid when the vehicle is under automatic control. All this information should be either state or continuously updated data, so all lines in the diagram represent data flow. The controller is implemented as a continuously-evaluating function that matches the dataflow character of the inputs and outputs. Several implementations are possible, including variations on simple on/off control, proportional control, and more sophisticated disciplines. Each of these has a parameter that controls how quickly and tightly the control tracks the set point; analysis of these characteristics is discussed in Section 3.4. As noted above, the engine is of little interest here; it might very well be implemented as an object or as a collection of objects.

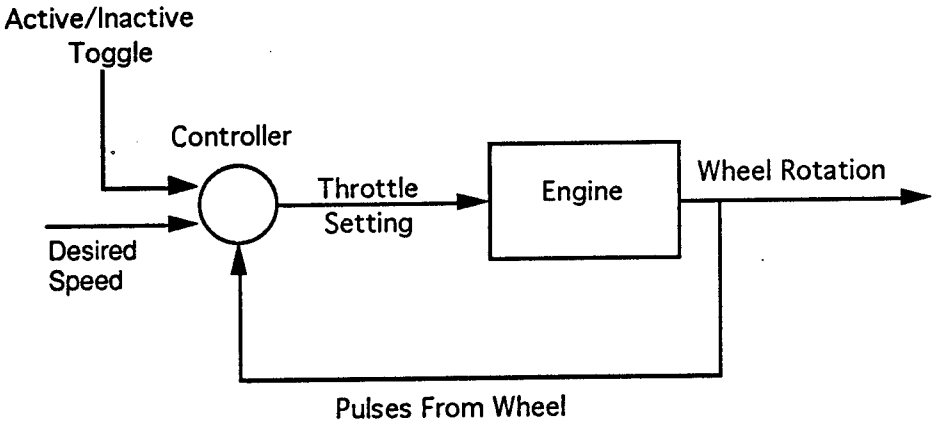


Figure 7: Control Architecture for Cruise Control

The set point calculation divides naturally into two parts: (a) determining whether or not the automatic system is active—in control of the throttle and (b) determining the **desired speed** for use by the controller in automatic mode.

Some of the inputs in the original problem definition capture state (**system on/off, engine on/off, accelerator, brake**) and others capture events (**wheel pulses, increase/decrease speed, resume, clock**). We will treat **accelerator** as state, specifically as a continuously-updated value. However, the determination of whether the automatic cruise control is actively controlling the car is cleaner if everything it depends on is of the same kind. We will therefore use transitions between states for **system on/off, engine on/off**, and **brake**. For simplicity we assume brake application is atomic so other events are blocked when the brake is on. A more detailed analysis of the system states would relax this assumption [Atlee and Gannon 91].

The active/inactive toggle is triggered by a variety of events, so a state transition design is natural. It's shown in Figure 8. The system is completely off whenever the engine is off. Otherwise there are three inactive and one active states. In the first inactive state no set point has been established. In the other two, the previous set point must be remembered: When the driver accelerates to a speed greater than the set point, the manual accelerator controls the throttle through a direct linkage (note that this is the only use of the accelerator position in this design,

and it relies on relative effect rather than absolute position); when the driver uses the brake the control system is inactivated until the resume signal is sent. The **active/inactive toggle** input of the control system is set to active exactly when this state machine is in state Active.

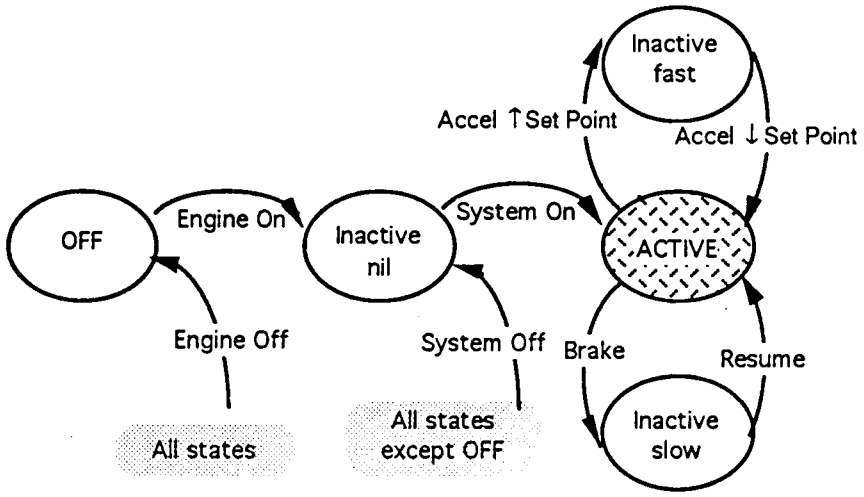


Figure 8: State Machine for Activation

Determining the **desired speed** is simpler, since it does not require state other than the current value of **desired speed** (the set point). Any time the system is off, the set point is undefined. Any time the **system on** signal is given (including when the system is already on) the set point is set to the current speed as modeled by **wheel pulses**. The driver also has a control that increases or decreases the set point by a set amount. This, too, can be invoked at any time (define arithmetic on undefined values to yield undefined values). Figure 9 summarizes the events involved in determining the set point. Note that this process requires access to the clock in order to estimate the current speed based on the pulses from the wheel.

Event	*Effect on desired speed*
Engine off, system off	Set to "undefined"
System on	Set to current speed as estimated from wheel pulses
Increase speed	Increment desired speed by constant
Decrease speed	Decrement desired speed by constant

Figure 9: Event Table for Determining Set Point

We can now (Figure 10) compose the control architecture, the state machine for activation, and the event table for determining the set point into an entire system. Although there is no need for the control unit and set point determination to use the same clock, we do so to minimize changes to the original problem statement. Then, since **current speed** is used in two components, it would be reasonable for the next elaboration of the design to encapsulate that model in a reusable object; this would encapsulate the clock.

All of the objects in Booch's design (Figure 6) have clear roles in the resulting system. It is entirely reasonable to look forward to a design strategy in which the control loop architecture is used for the system as a whole and a number of other architectures, including objects and state machines, are used in the elaborations of the elements of the control loop architecture.

The shift from an object-oriented view to a control view of the cruise control architecture raised a number of design questions that had previously been slighted: The separation of process from control concerns led to explicit choice of the control discipline. The limitations of the control model also became clear, including possible inaccuracies in the current speed model and incomplete control at high speed. The dataflow character of the model showed irregularities in the way the input was specified, for example mixture of state and event inputs and the inappropriateness of absolute position of the accelerator.

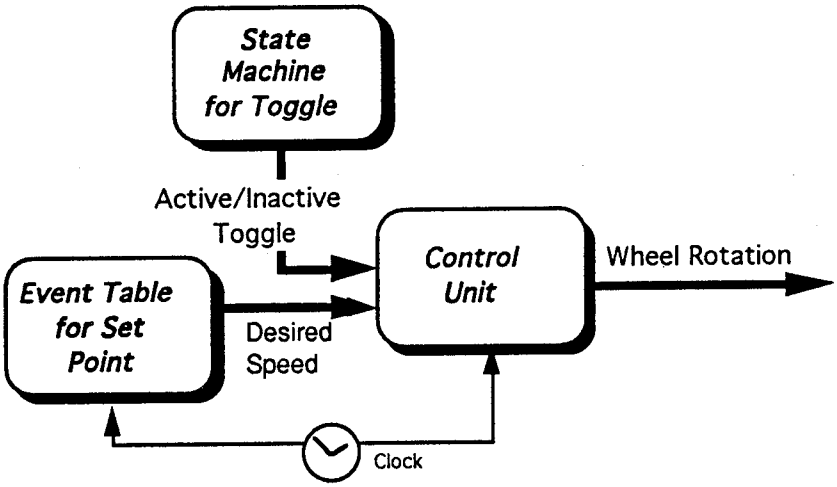


Figure 10: Complete cruise control system

3. Analysis and Discussion

3.1. Correspondence between architecture and problem

The selection of an architecture commits the designer to a particular view of a problem. Like any abstraction, this emphasizes some aspects of the problem and suppresses others. Booch [Booch 86] characterizes the views inherent in object-oriented and functional architectures:

> Simply stated, object-oriented development is an approach to software design in which the decomposition of a system is based upon the concept of an object. An object is an entity whose behavior is characterized by the actions that it suffers and that it requires of other objects. Object-oriented development is fundamentally different from traditional functional methods, for which the primary criteria for decomposition is that each module in the system represents a major step in the overall process.

The issue, of course, is deciding *which* abstractions are most useful for any particular problem. We have argued that the control view is particularly appropriate for a certain class of problems. In this case, the control view clarifies the design in several ways:

- The control view leads us to respecify the output as the actual speed of the vehicle.
- The separation of control from process makes the model of actual speed explicit and hence more likely to be validated; similarly it raises the question of control authority.
- The explicit element for the control algorithm also sets up a design decision about the kind of control to be exercised (see Section 3.4).
- By establishing special relations among components, the control paradigm discriminates among different kinds of inputs and makes the feedback loop more obvious.
- The control paradigm clearly separates manual operation from automatic operation.
- Determination of the set point is easier to verify when it's separated from control; for example, Booch's design does not appear to reset the desired speed to undefined when the engine is turned off.

The idea of using software for control is not new; scheduling algorithms and real-time operating systems are an established area. However, the control character of the task is rarely obvious from the system architecture. For example, the architecture of one commercial process control system [Shaw 91, Garlan and Shaw 93] is primarily a layered architecture in which the control elements at the control loop level appear as objects; here the separation between task and control is not obvious. The Chimera framework for sensor-based control systems [Stewart et al 92, Stewart et al 93] provides reusable, reconfigurable building blocks for control that interact via data flow, but that framework does not make the explicit separation between the controller and the process that we make here. The ARPA domain-specific software community is exploring canonical architectures for particular problem domains [Tracz 93]. Early results from this work include software architectures that quite appropriately include contain control loops. A future task is consideration of those designs in light of this formulation of the control loop idiom.

Contrast these cases with Booch's second example, the sea buoy. This requires continuing operations and state changes in response to external commands. However, it does not have to compensate for external perturbations and so is not a candidate for a control loop organization.

3.2. Language support

At some level the architecture of a system is independent of the programming language. However, architectures assume certain kinds of interactions among components, and these interactions often induce assumptions about the way the components are packaged -- that is, about some aspects of their interfaces. Unfortunately, programming language do not provide comparable levels of direct support to all architectural idioms. Some architectures, most notably procedures, objects, and processes, have motivated features in programming languages intended to support the structural needs of the architectures. Others, however are orphans

Booch remarks on the effect languages have on architectural design, simply because of the kinds of building blocks they present.:

> "Well-structured systems developed with older [i.e., imperative --MS] languages tend to consist of collections of subprograms (or their equivalent), mainly because that is structurally the only major building block available. Thus, these languages are best suited to functional decomposition techniques, which concentrate upon the algorithmic abstractions. But as Guttag observes, 'unfortunately, the nature of the abstractions that may be conve-

niently achieved through the use of subroutines is limited. Subroutines, while well suited to the description of abstract events (operations), are not particularly well suited to the description of abstract objects. This is a serious drawback' [Guttag et al 78]"

Since almost no languages make provisions for providing data flow interfaces, its small wonder that data flow architectures don't leap immediately to mind. In order to achieve data flow, you have to *want* to achieve data flow. For example, unix pipe and filter architectures assume communication via pipes. The individual modules achieve this with conventions about how read and write procedures are used.

3.3. Methodological implications

Object-oriented architectures are supported by associated methodologies. What can we say about methodologies for control-loop organizations and when they are useful? First, a methodology should help the designer decide when the architecture is appropriate. This is discussed in Sections 1.2 and 2.3. Second, a methodology should help the designer identify the elements of the design and their interactions. Such a list is established in Section 1.2 and exercised in Section 2.3; this corresponds to instructions for "finding the objects" in object-oriented methodologies. Third, a methodology should help the designer identify critical design decisions. In the case of control, these include potential safety problems as discussed in sections 2.3 and 3.5.

Åström and Wittenmark give a collection of examples of common solutions for process control problems [Åström and Wittenmark 84]. Each identifies a typical control situation and gives advice for suitable strategies. They also give a top-down methodology that also serves for the control paradigm for software:

- Choose the control principle
- Choose the control variables
- Choose the measured variables
- Create subsystems

A methodology should also provide for system modifications. Booch proposes two for the object-oriented design; both would be simple changes in the control-loop design:

- *Add a digital speedometer:* The wheel pulses are directly available as a control signal; this can be picked up by any other component and used independently of the control paradigm. In addition, Section 2.3 suggested creation of a object for current speed.
- *Use separate microcomputers for current/desired speed and throttle:* The most likely assignment of function to multiple processors would put the control on one and engine-related operations on another. This corresponds directly to the design.

3.4. Performance: system response to control

Process control provides powerful tools for selecting and analyzing the response characteristics of systems. For example, the cruise controller can set the throttle in several ways [Perry 84]:

- *On/Off Control:* The simplest and most common mode of control simply turns the process off and on. This is more appropriate for thermostats than throttles, but it could be considered. In order to prevent the power from fluttering rapidly on and

off, on/off control usually provides some form of hysteresis (actual speed must deviate from set point by some amount before control is exercised, or power setting can't be switched more often than a preset limit).

- *Proportional Control:* The output of a proportional controller is a fixed multiple of the measured error. The gain of a cruise controller is the amount by which the speed deviation is multiplied to determine the change in throttle setting. This is a parameter of control. Depending on the properties of the engine, this can lead to a steady-state value not quite equal to the set point or to oscillation of the speed about the set point.
- *Proportional plus Reset Control:* The controller has two parts, the first proportional to the error and the second to cause the controller output to change as long as an error is present. This has the effect of forcing the error to zero. Adding a further correction based on the derivative of the error speeds up the response but is probably overkill for the cruise control application.

For each of these alternatives, mathematical models of the system responses are well understood.

3.5. Correctness

When software controls a physical system, correctness and safety become particularly acute concerns. Sections 2.3 and 3.1 show how the control paradigm's methodology leads the designer to consider the accuracy of design assumptions that have significant safety implications. For cruise control, the possibility of runaway feedback is a significant safety concern, as the author's cruise control once vividly demonstrated.

4. Summary

Design methodologies get much of their power from focusing attention on significant decisions at appropriate times. They generally do this by decomposing the problem in such a way that development of the software structure proceeds hand-in-hand with the analysis for significant decisions. This localizes decisions and limits the ripples caused by changes. In this example we have explored an example in which the significant high-level decisions are better elicited by a methodology based on process control than on the more common object-oriented methodology.

Software architecture studies the ways software components are organized into systems, the reasons for selecting one architecture over another, and the analysis of the systems that result. Section 1.2 describes an architecture for control paradigms and provides a rule of thumb for deciding when to select this software organization:

> When the execution of a software system is affected by external disturbances—forces or events that are not directly visible to or controllable by the software—this is indication that a control paradigm should be considered for the software architecture.

The control paradigm separates the operation of the main process from compensation for external disturbances. This separation of concerns yields appropriate abstractions and leads to design questions that might otherwise be neglected.

Cruise control exemplifies a class of software system design problems in which a real-time process is controlled by embedded software. Conceptually such processes update the control status

continuously. Thinking about these designs explicitly as process control problems leads the designer to a software organization that separates process concerns from control concerns and requires explicit attention to the appropriateness and correctness of the control strategy. This leads to early consideration of performance and correctness questions that might not otherwise arise.

Acknowledgments

More than is usual, a clear sequence of events led to this paper. With David Garlan and other colleagues at CMU (both SCS and SEI), I've been studying architectural paradigms for several years; we have a regular reading group for current events in the area. Along the way I had the opportunity to learn about control software from colleagues at Fisher Controls. This spring Marc Graham arranged for most of the DARPA Domain-Specific Software Architecture groups to discuss their results with us. Will Tracz presented an architecture for avionics whose essential core was essentially a feedback loop. This provoked me to look again at process control to see how close the resemblance was, and a few days later while looking for a good example I realized that we've had one—cruise control—staring us in the face all along. Comments on the draft from Roy Weil, Jeannette Wing, Gene Rollins, and David Garlan have improved the result.

This research was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense) and by a grant from Siemens Corporate Research. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies of the U.S. Government, the Department of Defense, Siemens Corporation, or Carnegie Mellon University.

References

[Åström and Wittenmark 84] Karl J. Åström and Björn Wittenmark. *Computer-Controlled Systems.* Prentice-Hall 1984.

[Atlee and Gannon 91] Joanne Atlee and John Gannon. State-Based Model Checking of Event-Driven System Requirements. Proc. SIGSOFT Conference on Software for Critical Systems, *ACM Software Engineering Notes*, vol. 16, no. 5, December 1991.

[Booch 86] Grady Booch. Object-Oriented Development. *IEEE Trans. on Software Engineering* SE-12, 2, February 1986, pp. 211-221.

[Garlan and Shaw 93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds), *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific Publishing Company, 1993 (to appear).

[Guttag et al 78] John Guttag, Ellis Horowitz, and David Musser. The Design of Data Type Specification. *Current Trends in Programming Methodology* vol. 4, Prentice-Hall 1978, p.200.

[Leveson 86] Nancy Leveson. Software Safety: Why, What, and How. *ACM Computing Surveys*, vol. 18, no. 2, June 1986, pp.125-163.

[Perry 84] Robert H. Perry. *Perry's Chemical Engineer's Handbook.* McGraw-Hill 1984, Sec 22, Process Control.

[Seborg et al 89] Dale E. Seborg, Thomas F. Edgar, Duncan A. Mellichamp. *Process Dynamics and Control.* Wiley 1989.

[Shaw 91] Mary Shaw. Heterogeneous Design Idioms for Software Architecture. *Proc. Sixth International Workshop on Software Specification and Design*, IEEE, October 1991, pp.158-165.

[Shaw et al 94] Mary Shaw, David Garlan, Robert Allen, Dan Klein, John Ockerbloom, Curtis Scott and Marco Schumacher. *Candidate Model Problems in Software Architecture.* Unpublished manuscript.

[Stewart et al 92] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Integration of Real-Time Software Modules for Reconfigurable Sensor-Based Control Systems. *Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems*, July 1992, pp.325-333.

[Stewart et al 93] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. A Software Framework for Reconfigurable Robotic and Automation Systems. *CMU/RI Technical Report CMU-RI-TR-93-11*, May 1993.

[Tracz 93] Personal communication, 1993.

[Ward 84]. P. Ward used cruise control for an exercise at the Rocky Mountain Institute for Software Engineering, Aspen CO 1984; Booch adapted his formulation from Ward's.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890